Data Protector Encoding/Compression plug-in

Whitepaper

HP OpenView Storage

July 2006

CONTENTS:

INTRODUCTION **3**

OVERVIEW **4**

DATA PROTECTOR ENCODING/COMPRESSION PLUG-IN FEATURES 4
DATA PROTECTOR ENCODING/COMPRESSION PLUG-IN LIMITATIONS 4

DATA PROTECTOR ENCODING/COMPRESSION PLUG-IN API **6**

ENCODING API 6
COMPRESSION API 9
ERROR HANDLING 14
MESSAGES 14
INTEGRATING USER LIBRARIES WITH DATA PROTECTOR 16

WINDOWS DISASTER RECOVERY **25**

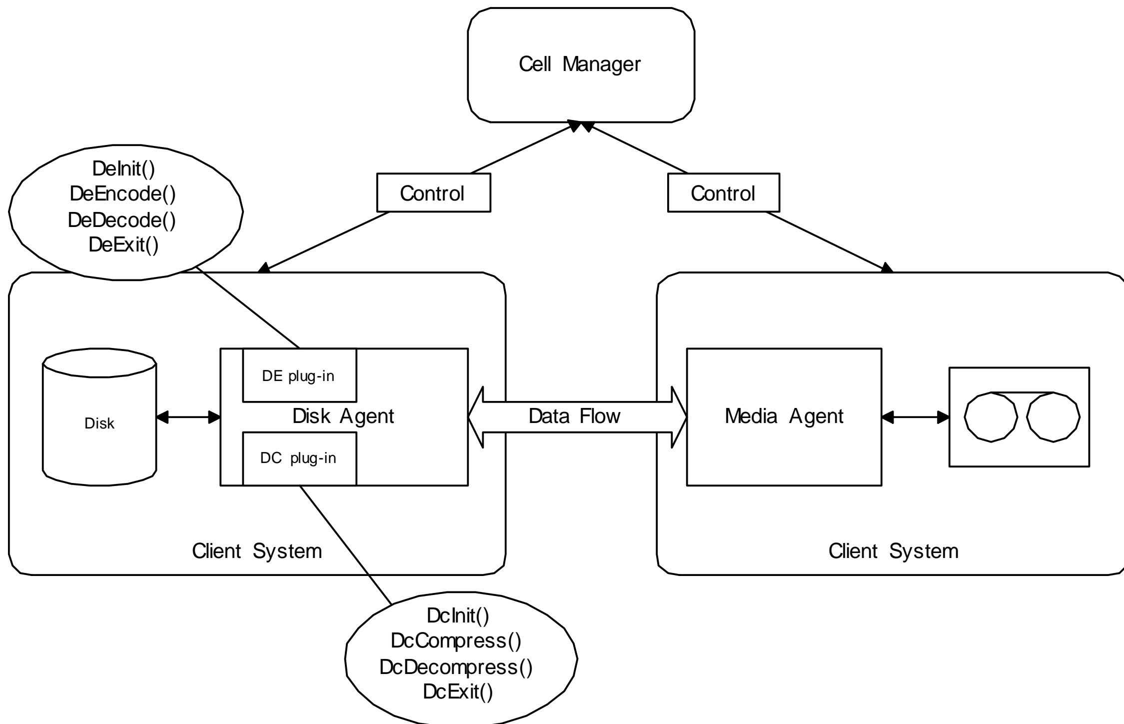
QUESTIONS AND ANSWERS **27**

APPENDIX 1 – TERMS AND DEFINITIONS **28**

Introduction

With A.06.00 Release, HP OpenView Storage Data Protector includes an enhanced version of encoding/compression plug-in, which supports implementation of user-specific compression and encoding algorithms instead of the default Data Protector compression/encoding algorithms.
The purpose of this document is to explain Data Protector's encoding/compression plug-in functionality and implications of its use to system and backup administrators. It also gives guidance to developers of applications and libraries for encoding/compression plug-in, on how to take a full advantage of this plug-in.
Check HP OpenView Storage Data Protector Concepts Guide and HP OpenView Storage Data Protector Administrator's Guide for general information on Data Protector. Refer to man pages on UNIX or HP OpenView Storage Data Protector Command Line Interface Reference for information on Data Protector commands.

Figure 1: The Data Protector Encoding/Compression plug-in



Overview

This chapter gives an overview of the Data Protector encoding/compression plug-in functionality and covers several key points for application developers as well as system/backup administrators using the encoding/compression plug-in.

Data Protector Encoding/Compression Plug-in Features

Data Protector currently supports encoding and compression features employing its own implementation. The default encoding algorithm is `XOR` while the default compression algorithm is Lempel-Ziv 4.3 (UNIX compress). It is possible to use these features using the `omnib` command or the GUI.
The Data Protector data compression and encoding logic is implemented as shared libraries[1] separated from the Disk Agent (DA). The Data Protector modules using the encoding/compression plug-in are:

- Volume Backup/Restore Disk Agent
- Data Protector Internal Database (IDB) Backup Disk Agent
- Raw Disk Backup/Restore Disk Agent
- Integration with HP OpenView OmniStorage
- Online Integrations

Data Protector Encoding/Compression Plug-in Limitations

- Features described in this document are limited to Data Protector A.06.00 and newer. See the HP OpenView Storage Data Protector Software Release Notes for information on the platforms on which the DA is supported. Encoding/compression plug-in not supported on HP NAS, OpenVMS and Windows Alpha platforms. Compression is not supported on Novell NetWare systems. However, it is still possible to decompress Novell NetWare files that were compressed during backup using Data Protector A.06.00 or older versions.
- The current version of Data Protector encoding/compression plug-in API is not fully compatible with previous versions.
  The libraries used with Data Protector A.05.00 cannot be used as is with Data Protector A.06.00 and vice versa. You have to modify and recompile your libraries using the API provided with Data Protector A.06.00 (described in this document).
  Backups, which were created with Data Protector A.05.00, can be restored using the Data Protector A.06.00 library, as long as the algorithm you used is the same as before.
  However, the default encoding/compression algorithm (using the libraries shipped with the product) is fully compatible with older versions of Data Protector and old backups.

[1] This includes Windows dlls.

- Data Protector encoding/compression plug-in does not provide the support for initialization/cleanup of user libraries. In case of an error in initialization functions, the Disk Agent will proceed without encoding/compression.
- It is not possible to dynamically change encoding/compression algorithms during a run-time depending on file system object type.
- User libraries must be redesigned to fit into Data Protector encoding/compression plug-in API.
- All errors in user libraries are treated as fatal, except for errors in initialization functions. In case of a fatal error, the Disk Agent will be immediately aborted, whereas in case of an error in initialization functions, the DA will proceed without encoding/compression.
- Sparse files are not supported.
- Software data compression for online database backups, such as Oracle, Sybase, SAP R/3, Informix, and Microsoft SQL, is not supported.

Data Protector Encoding/Compression Plug-in API

This chapter contains explanation of Data Protector encoding/compression plug-in API.

Encoding API

Initialization - DeInit()

Name:
`DeInit()` - Initializes Data Encoding Library

Synopsis:

UNIX
```
unsigned long DeInit(char *);
```

WINDOWS
```
__declspec(dllexport) unsigned long DeInit (TCHAR *);
```

NOVELL NETWARE
```
unsigned long HPDeInit(char *);
```

Parameters:
`char *` - a pointer to a 125 character long buffer.

Description:
The `DeInit` function is called by the DA module during the initialization procedure at startup. It is expected that a library, which uses this call, will perform all necessary initialization steps (obtaining key, internal initialization, etc.) and return the indication of what encoding will be used. The user custom library can put any identification string into this buffer. Length of the identification string is limited to 125 characters, including the terminating '`\0`' character. If the custom user library puts any string into this buffer, this string will be displayed with the appropriate Data Protector message, otherwise only Data Protector message indicating usage of the custom user library will be displayed.

Return Values:
`0` - Initialization failed, encoding disabled
`1` - XOR encoding
`[50-200]` - User-defined encoding types

Range 50-200 is reserved for user defined encoding types. Usage of values outside of the specified range can result in undefined behavior.

During the restore, return values will be used to check for the encoding type.

Error Handling:
Initialization failure during the backup will be reported by an error message on the console. DA will proceed *without* encoding the data.

Encoding - DeEncode()

Name:
`DeEncode()` - Performs data encoding.

Synopsis:

UNIX
```
int DeEncode(unsigned char *data, unsigned long size);
```

WINDOWS
```
__declspec(dllexport)
int DeEncode (unsigned char *data, unsigned long size);
```

NOVEL NETWARE
```
int HPDeEncode(unsigned char *data, unsigned long size);
```

Parameters:
`data` - A pointer to a raw data buffer;
`size` - Data buffer size

Description:
`DeEncode(…)` performs encoding of the data buffer. It is expected that encoding will be done in-place and that the data buffer size will be maintained while its contents will be overwritten by the encoded data.

Return Values:
`0` - Error in encoding process
`1` - Data encoded successfully.

Error Handling:
All errors will be treated as fatal and DA will be aborted.

Decoding - DeDecode()

Name:
`DeDecode()` - Decode data;.

Synopsis:

UNIX
```
int DeDecode(unsigned char *data, unsigned long size);
```

WINDOWS
```
__declspec(dllexport)
```

```
int DeDecode (unsigned char *data, unsigned long size);
```

NOVEL NETWARE

```
int HPDeDecode(unsigned char *data, unsigned long size);
```

Parameters:

`data` - A pointer to a raw data buffer;
`size` - Size of the data buffer

Description:

`DeDecode(…)` performs decoding of the raw data buffer. It is expected that decoding will be done in-place and that the data buffer size will be maintained while its contents will be overwritten by the decoded text.

Return Values:

`0` - Error in decoding process
`1` - Data decoded successfully.

Error Handling:

All errors will be treated as fatal and DA will be aborted.

Cleanup - DeExit()

Name:

`DeExit()` - cleanup process

Synopsis:

UNIX

```
int DeExit(void);
```

WINDOWS

```
__declspec(dllexport) int DeExit (void);
```

NOVEL NETWARE

```
int HPDeExit(void);
```

Parameters:

Description:

`DeExit()` is called to allow post-process cleanup.

Return Values:

`0` - Error
`1` - OK

Error Handling:

Return values of this function are ignored.

Callback functions

Data Protector provides two callback functions to read and write data. Pointers to these callback functions are passed as arguments to `DcCompress()` and `DcDecompress()` functions. These callback functions should be used for all read/write requests from the library.

dcCallback();

Name:
`dcCallback()` - reads from the file system or writes to the file system the object being backed up.

Synopsis:

```
typedef int (*dcCallback) (unsigned char *, unsigned long );
```

Parameters:

Description:
Callback functions are provided in order to enable read/write from user libraries.
Data Protector will pass pointers to callback functions to `DcCompress()` and `DcDecompress()` functions.

Return Values:
Since `dcCallback` function is used as a prototype for read/write, its return values depend on the context.

`readCallback`:

- `>0` - Number of bytes read
- `0` - Error or EOF

`writeCallback:`

- `>0` - number of bytes written
- `0` - Error

Notes:
The `readCallback` function returns the number of bytes read. This number can, under certain circumstances, be less than the number of bytes actually requested. The compression engine should not treat this as an error but simply issue a second read (this is similar to BSD sockets, reads may succeed partially). The `readCallback` function returns zero in case of error or at EOF.

The `writeCallback` function always succeeds completely. If an error condition is encountered, `writeCallback` never returns.

Initialization - DcInit()

Name:
`DcInit()` - Initializes the Compression Library.

Synopsis:

UNIX
```
unsigned long DcInit(char *);
```

WINDOWS
```
__declspec(dllexport) unsigned long DcInit (TCHAR *);
```

NOVEL NETWARE
```
unsigned long HPDcInit(char *);
```

Parameters:
`char *` - a pointer to a 125 character long buffer.

Description:
This function is called by the DA initialization procedure at startup. It is expected that the library will perform all initialization steps through this call and return an indication of what compression type will be used. User library can put any identification string into the passed buffer. Length of the identification string is limited to 125 characters including the terminating '`\0`' character. If the user library puts any string into this buffer, the string will be displayed with an appropriate Data Protector message, otherwise only Data Protector message indicating the usage of custom user library will be displayed.
During the restore, return value will be used to check for the type of compression.

Return Values:
`0` - Initialization failed, compression disabled.
`0x0b0b0b0b` – Lempel-Ziv 4.3
`[50-200]` - User-defined encoding types

Range 50-200 is reserved for user defined encoding types. Usage of values outside of the specified range can result in undefined behavior.

Error Handling:
Initialization failure during the backup will be reported by an error message on the console. DA will proceed *without* encoding the data.

Compression - DcCompress()

Name:
`DcCompress()` - Compress Data.

Synopsis:

UNIX
```
int DcCompress(dcCallback readCallback, dcCallback writeCallback);
```

WINDOWS

```
__declspec(dllexport)
int DcCompress (dcCallback  readCallback, dcCallback writeCallback);
```

Parameters:

`readCallback` - A pointer to the data input callback
`writeCallback` - A pointer to the data output callback

Description:

This is the actual stream compression function. It receives input data using the `readCallback` data input method and writes output using the `writeCallback` output functions. These callback functions are provided by Data Protector.
The nature of the input and output media is unknown to the compression module. Both input and output should be treated as read-only and write-only streams respectively.
The `DcCompress` function is called once for each file to be compressed. It should read its input until EOF (the input handler returns zero) and make sure all intermediate output buffers are flushed before it returns.
Each call to the input method is sooner or later directly translated into a `read()` system call. Thus, for better performance, the compression engine should try to read its input in large chunks and store them in an internal input data buffer.

Return Values:

`0` - Error during compression
`1` - Compression successful
All errors will be treated as fatal and DA will be aborted.

Notes:

Never call the readCallback to read zero bytes, as you will be returned zero, which means 'Error or EOF'. Also for the same reason, never attempt to write zero bytes.

Decompression - DcDecompress ()

Name:

`DcDecompress()` – Decompress data

Synopsis:

UNIX

```
int DcDecompress(dcCallback readCallback, dcCallback writeCallback);
```

WINDOWS

```
__declspec(dllexport)
int DcDecompress (dcCallback readCallback, dcCallback writeCallback);
```

NOVEL NETWARE

```
int HPDcDecompress(dcCallback readCallback, dcCallback writeCallback);
```

Parameters:

`readCallback` - A pointer to the data input callback
`writeCallback` - A pointer to the data output callback

Description:
This is the actual stream decompression function. It receives its input data using the `readCallback` data input method and writes its output using the `writeCallback` output functions.
Here are a few guidelines the compression module developer should keep in mind:

- The nature of the input and output media is unknown to the decompression module. Both input and output should be treated as read-only and write-only streams, respectively.
- The `DcDecompress()` function is called once for each file to be decompressed. It should read its input (until EOF the input handler returns zero) and make sure all intermediate output buffers are flushed before it returns.
- Both input and output may be done in byte chunks of arbitrary length. The `readCallback` typically returns data from an internal Data Protector data buffer. To increase performance, output larger amounts of data at once, because calls to the `writeCallback` may be directly translated into the `write()` system calls.
- The `readCallback` may succeed partially. A return value of zero denotes an error or the end of the current file to be decompressed. The `writeCallback` always succeeds completely and never reports any errors.

`readCallback` returns:
`>0` - Number of bytes read
`0` - Error or EOF
`writeCallback returns:`
`>0` - Number of bytes written
`0` - Error

Return Values:
`0` - Error
`1` - OK

Error Handling:
All errors will be treated as fatal and DA will be aborted.

Notes:
Never call the `readCallback` to read zero bytes, as you will be returned zero, which means 'Error or EOF'. Also for the same reason, never attempt to write zero bytes.

Cleanup - DcExit ()

Name:
`DcExit()` – Post-process cleanup

Synopsis:

UNIX
```
int DcExit (void);
```

WINDOWS
```
__declspec(dllexport) int DcExit (void);
```

NOVEL NETWARE

```
int HPDcExit(void);
```

Parameters:

Description:
This function allows DC module to perform post-process cleanup.

Return Values:
`0` - Error
`1` - OK

Error Handling

Data Protector does not provide support for any kind of recovery in case of errors in user library, so all errors in user libraries will be treated as fatal and DA will be aborted immediately. In case of initialization failure, DA will proceed without encoding/compression.

Messages

In the backup session reports, user (encoding/compression) library ID will be shown only if `encode` or `compress` is selected (in GUI or CLI).
During the restore process, messages indicating library initialization failure will be shown even if data was not encoded or compressed.

Encoding Related

```
[Normal] From: VBDA@fa.hermes.si "msg_test"  Time: 09/03/02 10:15:46
       <identification name> library used for encoding![2]
```

This message will be shown only if library initialization passes, otherwise the following message will be reported on the console:

```
[Warning] From: VBDA@colorado.hermes.si "DEDC"  Time: 03/07/03 16:13:06
        Encoding library initialization failed!
```

If the encode or decode process fails, the following message will be displayed:

```
[Critical] From: VRDA@fa.hermes.si "msg_test"  Time: 09/03/02 10:15:46
              Internal error in encoding library. Aborting session!
```

If encoding type is not recognized during restore, the following error will be displayed:

```
[Warning] From: VRDA@fa.hermes.si "msg_test"  Time: 09/03/02 10:49:10
[84:119]        Unknown type of encoding !
```

Compression Related

```
[Normal] From: VBDA@fa.hermes.si "msg_test"  Time: 09/03/02 10:15:46
   <identification name> library used for compression!
```

```
[Warning] From: VRDA@colorado.hermes.si "DEDC"  Time: 03/07/03 16:49:53
        Compression library initialization failed!
```

If compress or decompress process fails, the following message will be displayed:

[2] If the user library doesn't return the identification string, Data Protector will display the same message replacing `<identification name>` part with the string "`Unknown`". The same goes for compression.

```
[Critical] From: VRDA@fa.hermes.si "msg_test"  Time: 09/03/02 10:15:46
        Internal error in compression library. Aborting session!
```

If compression type is not recognized during restore, the following error will be displayed:

```
[Warning] From: VRDA@fa.hermes.si "msg_test"  Time: 09/03/02 10:15:46
[84:114]        Unknown type of compression: 185273099.


[Warning] From: VRDA@fa.hermes.si "msg_test"  Time: 09/03/02 10:32:17
                Invalid compress data !
```

Integrating User Libraries with Data Protector

Rationale

Data Protector encoding/compression plug-in enables users to embed their own encoding/compression libraries into the Data Protector modules. We assume that a user already has his own encoding/compression libraries[3] and would like to embed them into Data Protector, therefore significant changes in user libraries are not necessary.

Creating Libraries

Steps to create user libraries are explained. Process of creating shared libraries is not covered.

Encoding:

1. Declare encoding functions conforming DE encoding/compression plug-in API;
2. Implement those functions so they call the appropriate user's library functions in the following way:
   All initialization tasks should be done in `DeInit();`
   Use `DeEncode()` and `DeDecode()` to perform encoding and decoding;
   Use `DeExit()` to perform post-process cleanup;
3. Build new binaries linking with already existing library.[4] Note that Windows libraries must support UNICODE.
4. Create new shared library. It must me named `libde.sl` ;[5]

Compression:

1. Declare compression functions conforming DE encoding/compression plug-in API;
2. Implement those functions so they call appropriate user's library functions in the following way:
   All initialization tasks should be done in `DcInit();`
   Use `DcCompress()` and `DcDecompress()` to perform compression and decompression;
   Use `DcExit()` to perform post-process cleanup;
3. Build new binaries linking with already existing library. Note that Windows libraries must support UNICODE.
4. Create a new shared library. It must be named `libdc.sl` ;[6]

Configuration

Replace the Data Protector native library `libde.sl` (`libdc.sl`) with the newly built user library. It is recommended to keep the original libraries in a safe place.

---

[3] This assumption also covers writing the library from scratch, because a developer can easily design it to be compatible with Data Protector encoding/compression plug-in API.
[4] Data Protector does not provide separate binaries for 64 bit platforms, therefore binaries must be portable. On most platforms this can be done specifying a compiler option.
[5] This is HP-UX specific naming convention and it will be used in the rest of the document. Shared library extensions on different platforms are: AIX – `libde.a`, `libde.o`; IRIX, Solaris, Linux, SCO, SCO-5 - `libde.so`; Windows – `libde.dll`, MPE – `libde.sl`
[6] This is HP-UX specific naming convention and it will be used in the rest of the document. Shared library extensions on different platforms are: AIX – `libdc.a`, IRIX, Solaris, Linux, SCO, SCO-5- `libdc.so`; Windows - `libdc.dll`., MPE – `libdc.sl`

Novell Netware Specific

In order to support Encoding/Compression plug-in on Novell NetWare platforms, Library NLMs (named `HPLIBDE.NLM` and `HPLIBDC.NLM`) are provided.
Library NLMs are automatically loaded by Data Protector DA main modules (`HPVBDA.NLM` or `HPVRDA.NLM`) when DA is first loaded. Once the Library NLMs are loaded, all symbols are exported and any client NLM (Data Protector NLM) can use them. Once the library NLMs are loaded, they should not be loaded any more. Library NLMs will be unloaded only when the server goes down.
If you would like to uninstall Data Protector, manually unload library NLMs before uninstalling Data Protector. To manually unload library NLMs, unload `HPINET.NLM` first, then `HPLIBDC.NLM/HPLIBDE.NLM` libraries and finally the `HPBRAND.NLM` library.
`HPLIBDC.NLM` and `HPLIBDE.NLM` libraries are relatively small and should not take up too much server resources or affect server performance in any way.

Note that library NLMs can cause some unique problems. For example, if a library NLM allocates resources (such as memory) on behalf of another NLM, the library NLM needs to know when that another NLM is unloaded or otherwise terminated so that it can free the allocated resources. Refer to Novell NetWare development documentation for more information on library NLMs problems.

Encoding/Compression Library Templates

Source code templates for encoding/compression libraries are provided here. These examples provide instructions on how to build and create encoding/compression libraries on HP-UX systems. On Windows systems, data encoding/compression libraries are ordinary DLL files.

These libraries must support Unicode; in other words, your project settings should contain `_UNICODE` preprocessor definition. For more information about Unicode features, `TCHAR` type and `_T()` macro, check MSDN documentation. For any other platform, check compiler/linker documentation.

Encoding

```
/*----------------------------------------------------------------------
| encode.h
|
|                    Interface declaration
 ----------------------------------------------------------------------*/

#if _WIN32
__declspec(dllexport)
unsigned int DeInit(TCHAR *);
#else
unsigned int DeInit(char *);
#endif

#if _WIN32
__declspec(dllexport)
#endif
int DeEncode(unsigned char *data, unsigned long size);


#if _WIN32
__declspec(dllexport)
```

```
#endif
int DeDecode(unsigned char *data, unsigned long size);


#if _WIN32
__declspec(dllexport)
#endif
int DeExit(void);

/*---------------------------------------------------------------------
| encode.c
|
| DP Encode/Compress Plug-In Sample Code
| The sample code can be used only as an example on how to write Data
| Protector Encoding/Compression libraries. You should customize it
| based on your requirements and existing code base.
 --------------------------------------------------------------------*/

/*---------------------------------------------------------------------
| Compile Your code:
| cc +DAportable -c +z encode.c
|
| Create library
| ld -b -o libdemo.sl encode.o
 --------------------------------------------------------------------*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#ifdef _WIN32
#include <tchar.h>
#include <windows.h>
#else
  #include <unistd.h>
#endif

unsigned int EncodeType=114;

#include "encode.h"


/*---------------------------------------------------------------------
| Current implementation of Data Protector Encoding/Compression plug-in
| allows string identification of encoding/compression libraries as a
| message displayed in the Data Protector console.
| Note that this identification is just an addition to the encode type,
| which must be returned by the DeInit() call and therefore cannot be
| used instead of encoding type.
 --------------------------------------------------------------------*/

/*---------------------------------------------------------------------
| Let's use feature mentioned above to provide user friendly
| association of libraries we use.
 --------------------------------------------------------------------*/
```

```
#ifdef _WIN32
  const TCHAR *e_libId =_T("Encode Demo - Ver 1.2");
#else
 const char *e_libId="Encode Demo - Ver 1.2";
#endif


/*------------------------------------ LIBRARY IMPLEMENTATION -----------
----------------*/


unsigned int DeInit(
#ifdef _WIN32
        TCHAR *libId
#else
          char *libId
#endif
      )
      {
      /* Perform any kind of initialization here*/


      /*Use provided buffer to put extra identification information.
      These information will be displayed on DP console*/


#ifdef _WIN32
        wcscpy(libId, e_libId);
#else
        strcpy(libId, e_libId);
#endif
/*----------------------------------------------------------------------
| It is assumed that library will return unique identifier of the used
| encoding algorithm upon successful initialization. So, do not forget
| to do it!!
 --------------------------------------------------------------------*/

      return(EncodeType);
}


int DeEncode(unsigned char *data, unsigned long size) {
      /*There is place where you should perform data encoding.
    Note that data size must not be changed (Check API Section) */

      return(1);
}


int DeDecode(unsigned char *data, unsigned long size) {
      /*There is place where you should perform data decoding.
    Note that data size must not be changed (Check API Section).*/

      return(1);
}
```

```
int DeExit(void) {
      /*This is the place where you should perform any cleanup*/

      return(0);
}
```

Compression

```
/*----------------------------------------------------------------------
| compress.h
|
|                    Interface declaration
 ---------------------------------------------------------------------*/

typedef int (*dcCallback) (unsigned char *, unsigned long );

#ifdef _WIN32
__declspec(dllexport)
unsigned int DcInit(TCHAR *);
#else
unsigned int DcInit(char *);
#endif


#ifdef _WIN32
__declspec(dllexport)
#endif
int DcCompress(dcCallback readCallback, dcCallback writeCallback);


#ifdef _WIN32
__declspec(dllexport)
#endif
int DcDecompress(dcCallback readCallback, dcCallback writeCallback);


#ifdef _WIN32
__declspec(dllexport)
#endif
int DcExit(void);


/*----------------------------------------------------------------------
| compress.c
|
| DP Encode/Compress Plug-In Sample Code
| The sample code can be used only as an example on how to write Data
| Protector Encoding/Compression libraries. You should customize it
| based on your requirements and existing code base.
 ---------------------------------------------------------------------*/
```

```
/*------------- HPUX Build Instruction----------------------
| Compile Your code:
| cc +DAportable -c +z compress.c
|
| Create library
| ld -b -o libdemo.sl compress.o
 ---------------------------------------------*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#ifdef _WIN32
    #include <tchar.h>
    #include <windows.h>
#else
    #include <unistd.h>
#endif

#include "compress.h"

/*---------------------------------------------------------------------
| Current implementation of Data Protector Encoding/Compression plug-in
allows string identification of
| encoding/compression libraries as a message displayed in the Data
Protector console.
| Note that this identification is just an addition to the compression
type, which must be
| returned by DcInit() call and therefore cannot be used instead of
compression type.
 --------------------------------------------------------------------*/

/*---------------------------------------------------------------------
| Let's use the feature mentioned above to provide user friendly
| association of libraries we use
 --------------------------------------------------------------------*/

#ifdef _WIN32
 const TCHAR *_libId   = _T("Compression Demo - Ver 0.5");
#else
 const char *_libId    = "Compression Demo - Ver 0.5";
#endif

/*---------------------------------------------------------------------
| Every compression algorithm should be represented by a unique number
in the allowed range. Check
| API section of Encoding/compression Whitepaper for information about
allowed values.
 --------------------------------------------------------------------*/

unsigned int const CompressType=75;


/*----------------------- LIBRARY IMPLEMENTATION ---------*/
```

```
unsigned int DcInit(
#ifdef _WIN32
        TCHAR *libId
#else
        char  *libId
#endif
      )
      {
      /* Perform any kind of initialization necessary to Your library*/


      /*Use provided buffer to put extra identification information.
      These information will be displayed in the Data Protector
console*/
#ifdef _WIN32

        wcscpy(libId,_libId);
#else
        strcpy(libId,_libId);
#endif

/*----------------------------------------------------------------------
| It is assumed that library will return unique identifier of the used
| compression algorithm upon successful initialization. So, do not
| forget to do it!!
 ---------------------------------------------------------------------*/

      return(CompressType);
}


int DcCompress(dcCallback readCallback, dcCallback writeCallback) {
      unsigned char *DataBuffer=NULL;
      unsigned long ReadSize=1024;
      unsigned long Read=0;
      unsigned long WriteSize=0;
      unsigned long Written=0;



      if(NULL == (DataBuffer=(unsigned char *)malloc(ReadSize *
sizeof(unsigned char)))) {
            return(0);
      }


      do {
            Read=readCallback(DataBuffer,ReadSize);
            if(0 == Read) {
                  /*Handle possible read errors*/
                  break;
            }
            /*It is time to compress data. This is the place where you
should to it*/
            /*Don't forget error handling*/
```

```
            /*...Some compression code or call to function performing
compression*/


            /*Write data back to Data Protector stream*/
            WriteSize=Read;
            if(0 != WriteSize) {
                  Written=writeCallback(DataBuffer,WriteSize);
                  if(0 == Written) {
                        /*Handle possible write errors*/

                        return(0);
                  }
            }     /*end write data*/
      }while(0 != Read);


    if(NULL != DataBuffer) {  /* Release buffer */
            free(DataBuffer);
      }

      return(1);
}


int DcDecompress(dcCallback readCallback, dcCallback writeCallback) {
      unsigned char *DataBuffer=NULL;
      unsigned long ReadSize=1024;
      unsigned long Read=0;
      unsigned long WriteSize=0;
      unsigned long Written=0;


      if(NULL == (DataBuffer=(unsigned char *)malloc(ReadSize *
sizeof(unsigned char)))) {
            return(0);
      }

      do {
            Read=readCallback(DataBuffer,ReadSize);
            /*It is time to decompress data. This is the place where
you should to it*/
            /*...Some compression code or call to function performing
compression*/
            /*Don't forget error handling*/


            /*Write data back to Data Protecotr stream*/
            WriteSize=Read;
            if(0 != WriteSize) {
                  Written=writeCallback(DataBuffer,WriteSize);
                  if(0 == Written) {
                        return(0);
                  }
            }     /*end write data*/
      }while(0 != Read);
```

```
	if(NULL != DataBuffer) {  /* Release buffer */
		free(DataBuffer);
	}
	return(1);
}


int DcExit(void) {
	/*This is the place where you should perform any cleanup*/

	return 0;
}
```

Useful tips to test your library

After you have created the library, it is recommended to test it:

1. Configure the newly created library as described in the previous chapter.
2. Start few backup/restore sessions on different files, for example regular and null-content files.
3. Change the size of these files and try to perform backup and restore.

Your libraries should be capable of handling any buffer size passed to functions performing encoding/decoding; including 1 byte.

Windows Disaster Recovery

Data Protector uses a collection of files packed in two (32-bit systems) or three (64-bit systems) disaster recovery cab files to perform a disaster recovery of Windows systems. The default encrypting and compressing libraries are also included in the disaster recovery cab file. This means that if a different encrypting/compressing library was used for backup, all files restored during the disaster recovery will not be decrypted/decompressed (because the default encrypting/compressing library will be used to decrypt/decompress the data) and therefore useless. In order to be able to successfully perform disaster recovery on Windows, you will have to replace the default Data Protector encrypting (`libde.dll`) or compressing (`libdc.dll`) library in the disaster recovery cab file with your own custom library with the same filename as the original encrypting/compressing library.

You can use the Data Protector `omnicab.exe` utility (stored in `<Data_Protector_Home>\bin\utilns` on a Windows Cell Manager) to unpack and pack the disaster recovery cab file. The `Omnicab.exe` utility is not installed on 64-bit systems by default; but you can copy it along with the `omnicab_ns.ini` file from a 32-bit system or Data Protector installation medium (it is compressed in the `\i386\autodr.cab` file).

When you are preparing a disaster recovery image (DR diskettes for Assisted Manual Disaster Recovery, ISO CD image for Enhanced Disaster Recovery, OBDR tape for One Button Disaster Recovery or ASR set for Automated System Recovery), Data Protector uses the disaster recovery cab files stored on the local machine to prepare the disaster recovery image. Disaster recovery cab files are stored in:

- `On 32-bit systems`:
  `<Data_Protector_Home>\Depot\Drsetup\disk1` and
  `<Data_Protector_Home>\Depot\Drsetup\disk2`
- `On 64-bit systems:`
  `<Data_Protector_Home>\Depot\Drsetup64\disk1`,
  `<Data_Protector_Home>\Depot\Drsetup64\disk2` and
  `<Data_Protector_Home>\Depot\Drsetup64\disk3`.

You have to replace the encrypting/compressing library only on those systems where the disaster recovery images are prepared, because Data Protector will always copy the disaster recovery cab files from the local system when preparing an image. You have to replace the encrypting/compressing library only once, because Data Protector always uses the same disaster recovery cab file from the local system to prepare a disaster recovery image.

Perform the following steps to replace the encrypting/compressing library in the disaster recovery cab file:

1. In the Windows command prompt, type: `set DPRoot= <Data_Protector_Home>`.
   Example: `set DPRoot="c:\program files\omniback`."
2. Type `set InstPath= <decompress_location>` where `<decompress_location>` determines where the contents of the disaster recovery cab files are to be extracted.
   Example: `set InstPath=c:\temp`.
3. Type set srcPath=`<compress_location>`. The `<compress_location>` argument determines the location of the files that are to be compressed in the disaster recovery cab file. This should be the same location as for decompressing files.

4. Type `omnicab.exe -path <inifile_path> -decompress` to decompress the content of the disaster recovery cab file to the `<decompress_location>` specified in step 2. The `<inifile_path>` argument is the location of the `omnicab_ns.ini` file. Example: `omnicab.exe -path "c:\program files\omniback\bin\utilns\omnicab_ns.ini" –decompress`.
5. Replace the libraries in `<decompress_location>`.
6. Type `omnicab.exe -path <inifile_path` to compress the files stored in the `<compress_location>` directory back to the disaster recovery cab file. Example: `omnicab.exe -path "c:\program files\omniback\bin\utilns\omnicab_ns.ini"`
7. For other settings see the `omnicab_ns.ini` file.

Questions and Answers

Q: How to perform version management in order to use different encoding/compression algorithms?
A: Values returned by the `DeInit()/DcInit()` calls are used to uniquely identify encoding/compression algorithms. At the same time, user libraries can use buffer passed to the `DeInit()/DcInit()` to put extra information on used encoding/compression algorithms. Note that this information is shown only in the Data Protector console.
Here is an example of using 3 different encoding algorithms:

1. Define 3 unique values in the allowed range, which will be returned by the `DeInit()`. Each of these unique values should represent a different encoding algorithm.
2. For each encoding algorithm, create a corresponding library as described in chapter *"Creating Libraries"*.
3. Depending on which encoding is being used, configure the appropriate library as described in chapter *"Configuration"*.
4. For each additional algorithm you are planning to use, configure the appropriate library as described in chapter *"Configuration"*.

The same procedure also applies to compression algorithms.

Q: Are sparse files supported?
A: No.

Q: Does Data Protector encoding/compression plug-in handle null-content files properly?
A: Yes.

Q: How does encoding/compression plug-in handle multistream files?
A: Windows NTFS file system supports files with alternate data streams. Data Protector A.06.00 also supports files with alternate data streams, which means that a multistream file is backed up and restored together with its alternate data streams. Although multistream files are supported, alternate data streams are not compressed or encoded since they are treated as file attributes. During a backup session, when compression and encoding options are selected, only the main stream, which contains files data, is compressed and encoded.

Q: Can both encoding and compression be performed together?
A: Yes. Encoding and compression libraries are independent. If both are used at the same time, encoding and compression will be performed.

Q: How does `OB2ENCODE omnirc` variable affect custom encoding/compression?
A: If `OB2ENCODE` is specified, it will override the `Encode` option in GUI (`-encode` option in CLI). The encoding/decoding algorithm used is determined by the library, which is installed on the client being backed up or restored. You must set the `OB2ENCODE omnirc` variable on the Cell Manager system, in order to control the behavior in this way.

Appendix 1 – Terms and Definitions

DA – Disk Agent
DE – Data Encoding
DC – Data Compression